Automated Strategy Invention for Confluence of Term Rewrite Systems

Liao Zhang[1,2] , **Fabian Mitterwallner**[1] , **Jan Jakubův**[3] and **Cezary Kaliszyk**[4,1]
[1]University of Innsbruck
[2]Shanghai Jiao Tong University
[3]Czech Technical University in Prague
[4]University of Melbourne
zhangliao714@gmail.com, fabian.mitterwallner@uibk.ac.at, {jakubuv, cezarykaliszyk}@gmail.com

Abstract

Term rewriting plays a crucial role in software verification and compiler optimization. With dozens of highly parameterizable techniques developed to prove various system properties, automatic term rewriting tools work in an extensive parameter space. This complexity exceeds human capacity for parameter selection, motivating an investigation into automated strategy invention. In this paper, we focus on confluence of term rewrite systems, and apply AI techniques to invent strategies for automatic confluence proving. Moreover, we randomly generate a large dataset to analyze confluence for term rewrite systems. We improve the state-of-the-art automatic confluence prover CSI: When equipped with our invented strategies, it surpasses its human-designed strategies both on the augmented dataset and on the original human-created benchmark dataset ARI-COPS, proving/disproving the confluence of several term rewrite systems for which no automated proofs were known before.

1 Introduction

Term rewriting studies substituting subterms of a formula with other terms [Baader and Nipkow, 1998], playing an important role in automated reasoning [Bachmair and Ganzinger, 1994], software verification [Meseguer, 2003], and compiler optimization [Willsey *et al.*, 2021]. Mathematicians have developed various techniques to analyze the properties of term rewrite systems (TRSs). However, many properties are undecidable [Baader and Nipkow, 1998], implying that no technique can consistently prove a particular property. To navigate this undecidability, modern term rewriting provers typically employ complicated strategies, incorporating wide arrays of rewriting analysis techniques, with the hope that one will be effective. Each technique often accompanies several flags to control its behavior. The diversity of techniques and their controlling flags result in a vast parameter space for modern automated term rewriting provers.

Manually optimizing strategies for undecidable problems is beyond human capacity given the extensive parameter space. This inspires us to apply AI techniques to search for appropriate strategies automatically. In this paper, we focus on confluence, an important property of term rewriting, and discuss automated strategy invention for the state-of-the-art confluence prover CSI [Nagele *et al.*, 2017]. We modify Grackle [Hůla and Jakubův, 2022], an automatic tool to generate a strategy portfolio, encoding strategies that require transformations and complex schedules such as parallelism.

Directly using a tool like Grackle to randomly generate parameters for CSI may produce unsound results. This is a unique challenge compared to previous applications of Grackle [Hůla and Jakubův, 2022; Aleksandrova *et al.*, 2024]. The solvers to which Grackle was previously applied always produce sound results, while CSI's users need to carefully specify their strategies to ensure soundness.

We also augment the human-built confluence problems database (ARI-COPS)[1], a representative benchmark for the annual confluence competition (CoCo)[2]. Before 2024, CoCo used the COPS database as the benchmark. An unpublished duplicate checker is executed to remove duplicated problems in COPS, resulting in the ARI-COPS database, which is used in CoCo 2024. As ARI-COPS has been created manually, it includes only 566 TRSs. They are of high quality, but the relatively small number is still inadequate for data-driven AI techniques that require large amounts of training data. To handle this problem, we generate a large number of TRSs randomly, but ensure that they are interesting enough to analyze. For this, we develop a procedure to confirm a relative balance in the number of TRSs most quickly solved by different confluence analysis techniques within the dataset.

We evaluate our strategy invention approach in ARI-COPS and the augmented dataset. On both of the datasets, the invented strategies surpass CSI's competition strategy. In particular, we prove (non-)confluence for several TRSs that have not been proved by any automatic confluence provers in the history of the CoCo competition.

As an example, our invented strategy is able to disprove confluence for the ARI-COPS problem `846.ari` (`991.trs` in COPS), never proved by any participant in CoCo. The key is the application of the redundant rule technique [Nagele *et al.*, 2015] with non-standard arguments. CSI's competition strategy performs `redundant`

[1]https://ari-cops.uibk.ac.at/

[2]https://project-coco.uibk.ac.at/

`-narrowfwd -narrowbwd -size 7` prior to performing non-confluence analysis. The flags `narrowfwd` and `narrowbwd` determine the categories of redundant rules to generate. Our tool automatically discovered that by changing the original redundant rule transformation to `redundant -development 6 -size 7`, we can prove this problem. A larger value for the flag `development` causes a larger number of development redundant rules to be added. We notice that the value six is crucial as small values below three are ineffective for `846.ari`. This is only one of the several TRSs which our new strategies can solve as discussed in the later sections.

The main reason why it is difficult to discover new proofs in CoCo, is because CSI's competition strategy developed rewriting experts is very complicated, for which a comprehensive explanation is presented in the technical appendix. For example the competition strategy includes the development redundant rule technique [Nagele *et al.*, 2015]. The original evaluation of it shows no improvement over other redundant rule techniques in COPS at that time. Thus, CSI's developers decided not to use it in the competition strategy. As COPS grows, it becomes helpful in some new TRSs such as `846.ari`. However, the default strategy has only slightly changed over the past years, and the development redundant rule technique has never been tried. One reason for this could be that choosing sound parameters is challenging even for rewriting experts. Meanwhile, competition strategy is highly complicated and has a prohibitively large configuration space both in the number of parameters and structures of the strategy itself. We leverage Grackle to do the tedious strategy search. It can automatically optimize the strategies better than experts as the dataset grows. Other rewriting tools do not discover the proof perhaps because they do not implement the essential techniques for solving the problems.

Contributions. First, to our best knowledge, our work is the first application of AI techniques to automatic confluence provers. We automatically generate a lot of strategies for the state-of-the-art confluence prover CSI and combine them as a unified strategy. Second, we carefully design the parameter search space for CSI to confirm the soundness of strategy invention. Third, we build a large dataset for confluence analysis, comprising randomly generated TRSs and problems in the ARI-COPS dataset. Finally, empirical results show that our strategy invention approach surpasses CSI's competition strategy both in ARI-COPS and the augmented datasets. Notably, we discover several proofs for (non-)confluence that have never been discovered by any automatic confluence provers in the annual confluence competition.

2 Background

2.1 Term Rewriting

We informally define some theoretical properties of term rewriting in this section, hoping to ease the understanding of the behavior underlining automatic confluence provers. A formal description can be found in the technical appendix.

We assume a disjoint set of *variable* symbols and a finite signature of *function* symbols. *Constants* are function symbols with zero arity. The set of *terms* is built up from variables and function symbols. The set of variables occurring in a term t is denoted by $Var(t)$. A term rewrite system (TRS) consists of a set of rewrite rules $l \to r$ where $l, r \in terms$, $l \notin variables$, and $Var(r) \subseteq Var(l)$. We write $t_1 \to^* t_n$ to denote $t_1 \to t_2 \to \ldots \to t_n$ where n can be one. A TRS is *confluent* if and only if $\forall s, t, u \in terms(s \to^* t \wedge s \to^* u \Rightarrow \exists v \in terms(t \to^* v \wedge u \to^* v))$. Consider the TRS of $\{f(g(x), h(x)) \to a, g(b) \to d, h(c) \to d\}$ [Gramlich, 1996]. It is not confluent since $f(d, h(b)) \leftarrow f(g(b), h(b)) \to a$, and no rules are applicable to $f(d, h(b))$ and a. A rewrite rule $l \to r$ is called *left-linear* if no variable occurs multiple times in l. A TRS is called left-linear if all its rules are left-linear. Left-linearity is crucial for confluence analysis since most existing confluence techniques only apply to such systems. In this paper, a term is called *complex* if it is neither a variable nor a constant.

2.2 CSI

CSI is one of the state-of-the-art automatic confluence provers that participates in CoCo. It ranked first in five categories of competitions in CoCo 2024. To show (non-)confluence of TRSs, CSI automatically executes a range of techniques, scheduled by a complicated configuration document written by experts in confluence analysis. Subsequently, CSI either outputs `YES`, `NO`, or `MAYBE` indicating confluence, non-confluence, or indetermination, respectively.

CSI implements many techniques applicable to the analysis of TRSs (many of them parametrized or transforming the system into one that can be analyzed by other techniques) and utilizes a complicated strategy language to control them. In CSI, these techniques are called *processors*. They are designed to prove the properties of TRSs, perform various transformations, and check the satisfiability of certain conditions. The strategy language can flexibly combine the execution of processors such as specifying parallel or sequential applications, disregarding unexpected results, assigning time limits, and designating repeated applications. The details of the strategy language are presented in the technical appendix.

Since the generated proofs are almost always large and difficult to check manually, CSI relies on an external certifier CeTA [Thiemann and Sternagel, 2009] to verify its proofs. To utilize CeTA, CSI outputs a certificate of its proof in the certification problem format [Sternagel and Thiemann, 2014]. Given a certificate, CeTA will either answer `CERTIFIED` or present a reason to reject it. Not all processors implemented in CSI are verifiable because CSI cannot produce certificates for all processors, and CeTA does not implement the verification procedures for all processors.

2.3 Grackle

Grackle [Hůla and Jakubův, 2022] is a strategy optimization system designed to automate the generation of various effective strategies for a given solver based on benchmark problems. Such solvers receive a problem and decide the satisfiability of a particular property of the problem. It was originally designed for automated reasoning tools and has been applied to various provers such as Prover9 [McCune, 2005] and Lash [Brown and Kaliszyk, 2022]. We choose Grackle for our research, as it is highly adaptable and we are not aware of

Algorithm 1 GrackleLoop: an outline of the strategy portfolio invention loop.

```
Input: initial strategies $\mathcal{S}$, benchmark problems $\mathcal{P}$,
       hyperparameters $\beta$
Output: a strategy portfolio $\Phi$
1: $\Phi_{strat} \leftarrow \mathcal{S}$
2: while termination criteria is not satisfied do
3:     Evaluate($\mathcal{P}, \Phi, \beta$)
4:     $\Phi_{cur} \leftarrow$ Reduce($\mathcal{P}, \Phi, \beta$)
5:     $s \leftarrow$ Select($\mathcal{P}, \Phi, \beta$)
6:     if $s$ is None then return $\Phi$
7:     $s_0 \leftarrow$ Specialize($s, \mathcal{P}, \Phi, \beta$)
8:     $\Phi_{strat} \leftarrow \Phi_{strat} \cup s_0$
9: end while
```

any strategy invention program that would allow the kinds of strategies needed for automatic rewriting tools. Additionally, Grackle has achieved good results with the solvers it was previously applied to. The strategy invention problem of Grackle is formally defined below.

Definition 1 (Strategy Invention Problem)**.** *Assume a set of initial strategies $\mathcal{S}$. In the benchmark of examples $\mathcal{P}$, the problem is to invent a bounded set of complementary strategies $\mathcal{S}'$ that can prove the largest number of problems in $\mathcal{P}$. Complementary strategies means that $\forall s'_i \in \mathcal{S}'$, s'_i should master a subset of problems $\mathcal{P}'_i \subseteq \mathcal{P}$, such that $\forall i \neq j$, $s'_j \in \mathcal{S}'$ cannot solve any problem in $\mathcal{P}'_i$ quicker than s'_i.*

Algorithm 1 outlines the strategy portfolio invention loop of Grackle, which invents strategies via a genetic algorithm and parameter tuning with randomness. The variable Φ denotes the current state, including information like all invented strategies Φ_{strat}, and the current generation of strategies Φ_{cur}. The first phase is *generation evaluation (evaluate)*. In this phase, Grackle evaluates all strategies Φ_{strat} in its portfolio on the benchmark $\mathcal{P}$. The evaluation results are stored in Φ to avoid duplicated execution.

Next, Grackle performs *generation reduction (reduce)*. It assigns scores to every strategy in Φ_{strat} based on the evaluation results in the previous phase. A configurable number of strategies with the highest scores becomes the current generation of strategies Φ_{cur}.

The third phase is *strategy selection (select)*. It selects a strategy s from the current generation of strategies Φ_{cur} based on certain criteria, which is then used to invent new strategies. If no strategy can be selected, the algorithm terminates.

Finally, *strategy specialization (specialize)* invents a new strategy s_0 via specializing s over its best-performing problems $\mathcal{P}_s$ in $\mathcal{P}$. Grackle then executes external parameter tuning programs such as ParamILS [Hutter *et al.*, 2009] or SMAC3 [Lindauer *et al.*, 2022], tuning parameters for the selected strategy s with randomness. The goal is to invent a new strategy s_0 such that it performs better than s ion $\mathcal{P}_s$. The new strategy s_0 will be added to the portfolio Φ.

Grackle employs the same approach to describe its parameter search space as ParamILS. The space is described by a set of available parameters, each of which is associated with a default value and several disjoint potential values. Grackle users need to input the potential values based on their domain-specific experiences on the particular solvers. We refer to [Hůla and Jakubův, 2022] for a comprehensive explanation of Grackle.

3 Strategy Invention and Combination

To generate a better strategy for CSI, we first invent a large set of complementary strategies, and then appropriately combine a subset of the invented strategies into a single strategy.

3.1 Strategy Invention

To find new strategies for CSI, we first need to represent the parameter space in a meaningful way. The parameter space needs to be designed with precision to guarantee soundness.

There are three reasons why CSI may produce unsound results given an entirely random strategy. First, some processors are not intended for confluence analysis. They may intend to prove other properties of TRSs, such as termination [Baader and Nipkow, 1998]. Second, even for the same processor, it may be designed to prove different properties of TRSs with different flags. Third, some transformation processors may change the goal of CSI to prove another property of TRSs, which is different from confluence such as relative termination [Zantema, 2004].

We separate CSI's competition strategy into 23 sub-strategies, which, along with CSI's competition strategy, also serve as the initial strategies for Grackle. Among the 23 sub-strategies, nine are mainly used to show confluence, and 14 are used to show non-confluence. A comprehensive explanation of the division is shown in the technical appendix.

We maintain the structure used in CSI's competition strategy during the strategy invention because CSI relies on certain combinations of processors to (dis)prove confluence. There are papers proving theorems for confluence analysis, stating that if some properties of a TRS can be proved, then it is (non-)confluent. Such a theorem can be implemented as a single processor, which checks whether the given TRS satisfies the properties required by the theorem. However, not all such theorems are implemented as a processor. To utilize such theorems, we need to combine CSI's strategy language and processors to perform transformations on the original TRS and prove the necessary properties of the transformed problem. If we generate strategies randomly, it will be difficult to generate such useful structures and may produce unsound strategies due to inappropriate transformations.

We search for three categories of parameters. First, we search for *processor flags* which do not violate the soundness guarantee. For instance, `-development 6` in Section 1 is a processor flag for the `redundant` processor. To ensure soundness, we only search for flags of processors existing in CSI's competition strategy. Second, we include *iteration parameters*, such as time limits or repeated numbers of execution, to regulate the running of a certain sub-strategy. These parameters are defined in CSI's strategy language. Moreover, we add a *boolean execution-controlling parameter* for some parallel or sequential executed sub-strategies, indicating whether to run the particular sub-strategies in confluence analysis. Assume a strategy `A||B`, where `||` denotes a parallel execution. The boolean parameters for `A` and `B` can represent whether to run one, both, or neither of them.

We need to construct a strategy for CSI using the parameters searched by Grackle. To achieve this, we start with CSI's competition strategy, replacing the processor flags and iteration parameters with relevant invented parameters. Then, we disable sub-strategies according to the boolean execution-controlling parameters.

The most challenging part of our work is the proper definition of the parameter space to confirm CSI's soundness. As the exact definition is quite technical and verbose, we present the explanation of the parameter space and show an invented strategy in the technical appendix.

3.2 Strategy Combination

After inventing several complementary strategies, we want to combine them into a single strategy and compare it with the competition strategy of CSI. The combination is performed by choosing a few strategies from Grackle's final portfolio and appropriately assigning a time limit to each of them.

To effectively divide the time, we split the whole one minute into several time splits. Next, we greedily allocate a strategy to each time split in the sequence by order. Each newly chosen strategy aims at proving the largest number of remaining benchmark problems that have not been proved by the previously chosen strategies. We shuffle the sequence 100 times and greedily select strategies for each shuffled sequence, resulting in strategy schedules comprising sequences of pairs of strategies and time splits. To use a strategy schedule, CSI executes each strategy in it by order for a duration of the relevant time split. We split the one-minute duration into many sequences and perform the greedy strategy selection for each. We finally choose the strategy schedule that maximizes the number of provable problems. The details of the strategy combination are explained in the technical appendix.

4 Dataset Augmentation

Although ARI-COPS is meticulously built by term rewriting experts, it is unsuitable for AI techniques. First, it is relatively small which is insufficient for contemporary AI techniques. Second, there may be an imbalance in ARI-COPS because the problems come from rewriting literature. The examples are often of theoretical interest and are constructed to illustrate specific confluence analysis techniques. However, TRSs encountered in practical applications can contain redundant rules that are irrelevant to illustrating a certain property.

4.1 TRS Generation Procedure

We develop a program to randomly generate a large dataset of TRSs, receiving multiple parameters to control the overall generation procedure. First, the maximum number of available function symbols F, constants C, variables V, and rules R establish the upper bound of the respective quantities of symbols and rules. For each of F, C, and V, a value is randomly selected between zero and the specified maximum, determining the actual number of available symbols. The actual number of rules is randomly chosen between one and R. Second, we define a parameter M, used during the initialization of function symbols. For each function symbol, an arity is randomly chosen between one and M

Another important parameter is the probability of generating a left-linear TRS L, which is associated with the likelihood of producing provably confluent TRSs. The majority of contemporary techniques for proving confluence are merely effective for left-linear TRSs. Without regulating the ratios of left-linearity, randomly generated TRSs rarely exhibit left-linearity, making it theoretically difficult to show confluence for them. We also notice that, in practice, CSI can merely prove confluence of very few generated TRSs if the ratios of left-linearity are not controlled. By default, we force 60% of generated TRSs to be left-linear.

Moreover, for a rule $l \to r$, there is a parameter called CT related to the probability of generating l and r that are complex terms. We need it because we prefer complex terms, whereas constants and variables are quite simple.

Algorithm 2 presents the generation procedure of a single term. While choosing the root symbol, we first randomly sample a value between zero and one and compare it with $comp$ to determine whether to only use $funs$ as candidates for the root symbol. Here, $comp$ is a value randomly chosen between zero and CT during the initialization stage of the generation of a TRS. If the $comp$ is larger than one, we can only generate complex terms. Meanwhile, according to the definition of rewrite rules in Section 2.1, the left term l in $l \to r$ cannot be a variable. After choosing a root symbol for the term t, we continuously choose new symbols for undefined function arguments until all of them are defined. After selecting a new variable, we need to remove it from the set of available variables if we are generating a left-linear TRS. The size of the terms generated by us is at most 15, where the *size* of a term is defined as the number of symbols in it. We choose 15 as the maximum value because the sizes of most terms in ARI-COPS are smaller than 15.

To generate a rule $l \to r$, we first execute Algorithm 2 to generate l and then execute it again to generate r. We extract all used variables in l and mark them as available variables for the generation of r, thereby $Var(r) \subseteq Var(l)$, as required by the definition of rewrite rules in Section 2.1.

We repeatedly generate rewrite rules until they reach the expected number and then return the newly generated TRS.

4.2 Dataset Generation

We utilize the program explained in this section to construct a large dataset, facilitating the application of AI techniques to confluence analysis. First, we randomly generate 100,000 TRSs with the parameters of the maximum number of available function symbols $F = 12$, constants $C = 5$, variables $V = 8$, and rules $R = 15$. Other parameters include the maximum arity of function symbols $M = 8$, the probability of generating left-linear TRSs $L = 0.6$, and the value related to the possibility of generating complex terms $CT = 1.6$.

However, the randomly generated dataset can be imbalanced. First, there may be significant differences in the number of confluent, non-confluent, and indeterminate TRSs. Second, the number of TRSs mastered by different confluence analysis techniques may vary considerably.

Algorithm 2 Term Generation

```
Input: $consts, vars, funs$
    $comp$, the likelihood of making a complex term
    $left$, whether the term is on the rewrite rule's left side
    $linear$, whether to construct a linear term
Output: a term $t$
 1: if $random(0,1) < comp$ then
 2:     $root\_symbols \leftarrow funs$
 3: else if left then
 4:     $root\_symbols \leftarrow funs + consts$
 5: else
 6:     $root\_symbols \leftarrow funs + consts + vars$
 7: end if
 8: $t \leftarrow random\_choose\_one(root\_symbols)$
 9: $undefs \leftarrow$ undefined function arguments in $t$
10: while $undefs$ is not empty do
11:     for all $undef \in undefs$ do
12:         $sym \leftarrow random\_choose\_one(funs + consts + vars)$
13:         replace the undefined function argument corresponding to $undef$ in $t$ with $sym$
14:         if $linear$ and $is\_var(sym)$ and $left$ then
15:             remove $sym$ from $vars$
16:         end if
17:     end for
18:     $undefs \leftarrow$ undefined function arguments of $t$
19: end while
20: return $t$
```

We develop a multi-step procedure to build a relatively balanced dataset. First, we execute CSI's competition strategy on all generated TRSs for one minute using a single CPU. CSI outputs NO, YES, and MAYBE for 69317, 25012, and 5671 TRSs, respectively.

Second, we randomly choose 5,000 problems from each set of problems classified as NO, YES, and MAYBE by CSI.

Third, we execute the duplicate checker used in CoCo 2024 to remove the duplications in the 15,000 chosen TRSs and 566 ARI-COPS TRSs. It checks the equivalence of syntactical structures between TRSs modulo renaming of variables and a special renaming on function symbols of their signatures. If TRSs of an equivalence class occur both in the randomly generated dataset and ARI-COPS, we only remove those randomly generated TRSs.

Fourth, we want to mitigate the imbalance in the number of problems mastered by different confluence techniques. We execute 26 strategies for all TRSs, aiming at labeling each of them with the most effective strategy. The labeling strategies contain all initial strategies for Grackle, which are explained in Section 3.1. The other two that are used to prove confluence are extracted from two complicated initial strategies, both consisting of many sub-strategies and integrated with transformation techniques that potentially simplify the search for proofs. Specifically, the two complicated initial strategies parallelly execute two important confluence analysis techniques, development closedness [Van Oostrom, 1997] and decreasing diagrams [Van Oostrom, 1994], not used by the other initial sub-strategies. If we do not use them for labeling, we will not be able to understand whether a TRS is mastered by one of the two important confluence analysis techniques. The details of the two new labeling strategies are explained in the technical appendix. The time limit for using CSI's competition strategy as a labeling strategy is one minute. The time limit for other labeling strategies is 30 seconds, smaller than one minute because the execution of decomposed sub-strategies is more efficient. We calculate the number of problems most quickly solved by each labeling strategy. The details of labeling strategies are presented in the technical appendix. The randomly generated dataset is quite imbalanced, four strategies master more than 1,000 problems; however, 16 strategies master less than 250 problems. To address the imbalance, we randomly choose at most 300 problems for a strategy from its set of mastered problems. We also randomly add 1,200 problems that cannot be solved by any labeling strategy to the dataset.

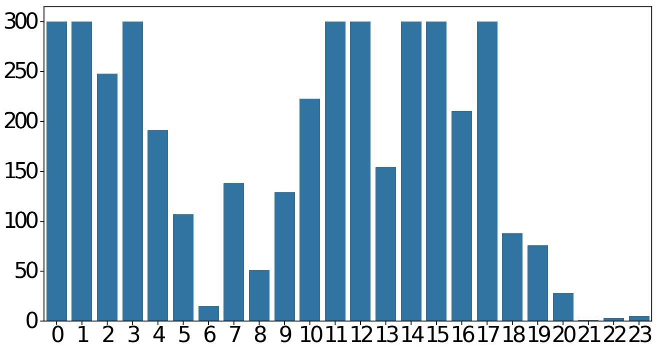

Figure 1: The number of TRSs solved most quickly (y-axis) for each labeling strategy (x-axis). Two labeling strategies that do not master any problems are ignored in the x-axis.

Finally, we obtain a dataset of 5,267 TRSs. Within this dataset, 1,647 TRSs are classified as confluent, 1,910 as non-confluent, and 1,710 as indeterminate when evaluated by CSI using a single CPU within a one-minute time limit.

Figure 1 shows the final distribution of the number of problems mastered by each labeling strategy. It is not perfectly balanced; however, we consider it relatively balanced, given that certain strategies can only master problems that satisfy particular properties. Such properties can be uncommon in randomly generated TRSs and practical applications.

There are infinitely many strategies that can be chosen as labeling strategies, such as strategies obtained by changing processor flags. We do not choose other labeling strategies as we have already decomposed CSI's competition strategy, enabling us to label problems with all categories of confluence analysis techniques implemented in CSI. Further decomposition or modification of processor flags may allocate problems to different labeling strategies that only slightly differ.

5 Experiments

We evaluate our strategy invention method on ARI-COPS and a combination of the randomly generated TRSs and ARI-COPS datasets. In both datasets, CSI with invented strategies outperforms CSI with the competition strategy, the state-of-the-art approach in confluence analysis for TRSs.

	ARI-COPS		augment	
CPU	1	4	1	4
init	475	477	846	852
total	479	484	873	871
confs	73	93	92	104
both in final	6	2	22	12

Table 1: Statistics of Grackle's training procedure. The rows *init* and *total* denote the number of problems solved by Grackle's initial strategy and the number of problems solved by strategies in Grackle's final portfolio, respectively. The row *confs* denotes the number of strategies that remains in Grackle's final portfolio. The row *both in final* represents the number of strategies in the final portfolio that master both confluence and non-confluence of TRSs.

5.1 Experimental Settings

The ARI-COPS 2024 dataset comprises a total of 1,613 problems of which 566 are TRS problems. We focus on evaluating our approach on TRS problems since they are standard term rewriting problems for confluence analysis and represent the major category in ARI-COPS. Another evaluation dataset consists of data from both ARI-COPS and our randomly generated datasets in Section 4.2. For training purposes, we arbitrarily select 283 examples from ARI-COPS and 800 examples from the randomly generated dataset. To build the test dataset, we exclude the examples in the training dataset, subsequently randomly selecting 800 examples from the randomly generated dataset and the remaining 283 examples from ARI-COPS.

The Grackle time limit for proving a TRS is 30 seconds, employed both in the evaluation and the strategy specialization phases. During the specialization phase, Grackle launches ParamILS for parameter tuning. The overall time limit for one strategy specialization phase is 45 minutes. The total execution time of Grackle is two days. Grackle performs parallel execution in both the evaluation and specialization phases; thus, we also limit the number of CPUs it can use. For each dataset, we perform two Grackle runs, configuring the numbers of available CPUs for a single strategy run to be either one or four. When it is set to one and four, the total number of available CPUs for Grackle is set to 52 and 66, respectively. Here, a CPU denotes a core of the AMD EPYC 7513 32-core processor. Grackle's portfolio stores at most 200 of the best strategies.

The use of four CPUs has been selected to match the results of CSI's competition strategy in CoCo 2024 on the competition setup. Given exactly the same problems solved by CSI in our own setup described above with four CPUs and in the CoCo competition in their Starexec [Stump *et al.*, 2014] setup we consider the further comparisons in the paper fair.

5.2 Experimental Results

Performance on ARI-COPS. Table 1 depicts the statistics of Grackle's training procedure. The value *total* shows the number of solved TRSs after the training, while *init* is the number solved by the initial strategies. When using four CPUs, Grackle's final portfolio contains more strategies than those in the final portfolio generated using one CPU. A probable reason is that executing with four CPUs can discover some strategies that are only effective with enough computation resources. The final augmented portfolios contain more strategies that master both confluence and non-confluence of TRSs. The likely reason is that a larger dataset makes training slower, and it is more difficult for Grackle to find optimal strategies for particular theoretical properties of TRSs.

	comp		total		combine		CoCo
CPU	1	4	1	4	1	4	
yes	266	272	271	277	271	276	272
no	203	205	208	207	207	207	205
solved	469	477	479	484	478	483	477

Table 2: Numbers of solved TRSs on ARI-COPS. The column *comp* represents CSI's competition strategy, *total* shows the total number of problems proved by all invented strategies, and *combine* denotes combining invented strategies as a single strategy. CoCo denotes the results obtained by CSI in CoCo 2024.

	never by CSI			never in CoCo		
CPU	yes	no	solved	yes	no	solved
1	2	3	5	1	3	4
4	4	2	6	1	2	3
1&4	6	3	9	2	3	5
1-CeTA	0	3	3	0	3	3
4-CeTA	1	0	1	0	0	0
1&4-CeTA	1	3	4	0	3	3

Table 3: Numbers of TRSs solved by all strategies in Grackle's final portfolio that have never been solved by all versions of CSI or any tool in CoCo. The suffix CeTA denotes the proofs can be certified by CeTA. The notion 1&4 means the union of all strategies invented by employing one CPU and four CPUs per strategy execution.

Table 2 compares the invented strategies with CSI's competition strategy. With a single CPU per each strategy evaluation, Grackle's final portfolio proves ten more problems than CSI's competition strategy. With four CPUs, *total* proves seven more problems than *comp*.

The invented strategies additionally (dis)prove several TRSs that have never been proved by different versions of CSI or all CoCo's participants, as depicted in Table 3. In total, we show (non-)confluence for nine TRSs that could not be solved by any versions of CSI. Five of the nine new proofs have never been proven by all CoCo's participants.

We combine the invented strategies as a single strategy to compare it with CSI's competition strategy. The number of time splits and the exact time assigned for each invented strategy are presented in the technical appendix. With single and four CPUs, *combine* proves nine and six more problems than the competition strategy, respectively.

When using one CPU, we gain more improvements over CSI's competition strategy compared to using four CPUs. A likely reason is that our strategy invention approach is particularly good at generating efficient strategies. With four CPUs, CSI can run several processors in parallelly, effectively reducing the runtime.

Certification. First, we check whether the answers found by the invented strategies are consistent with the answers discovered in CoCo. Second, we execute CeTA to verify the

	comp		combine	
CPU	1	4	1	4
yes	403	412	412	418
no	399	442	450	449
solved	802	854	862	867

Table 4: Numbers of solved TRSs on the testing examples of the augmented dataset.

proofs for the newly solved problems. Table 3 depicts the number of newly solved problems certifiable by CeTA. If we cannot certify the proofs due to the limitation of CeTA and CSI as explained in Section 2.2, we analyze the related strategies. We aim to understand what changes they perform to the original strategy lead to the proofs. From the analysis, we either slightly modify the sub-strategy defined in the competition strategy or directly use some existing sub-strategies to produce the same answers as the invented strategies. These modifications that lead to the answers are employed in the corresponding invented strategies, which are small and sound according to our knowledge of term rewriting. We also check the certification errors output by CeTA to figure out whether they are indeed errors or just caused by limitations of CSI and CeTA. Third, for each strategy in Grackle's final portfolio, we run CSI on its mastered problems and apply CeTA to verify the proofs. Only 234 and 226 proofs can be verified when one and four CPUs are employed for strategy invention, respectively. We manually check the proofs that cannot be verified by CeTA. The details of our certification procedures are shown in the technical appendix.

Performance on the augmented dataset. Table 1 also summarizes Grackle's training procedure in the augmented dataset. Compared to the training in ARI-COPS, Grackle's final portfolios consist of more strategies. The likely reason is that the augmentation dataset comprises more examples, necessitating more diverse strategies to cover them. We notice that with one CPU, the invented strategies prove more problems than those invented with four CPUs. This is probably caused by the randomness in the strategy invention.

The results of the evaluation in the test dataset are presented in Table 4. With one and four CPUs, *combine* respectively proves 60 and 13 more problems than *comp*. Notice that here the training examples are disjoint from the testing examples, whereas in the evaluation for ARI-COPS, they are the same. From this, we can conclude that our invented strategies generalize well to unseen data. With four CPUs, the unified strategy proves more problems than using one CPU. The likely reason is that the invented strategies with four CPUs can discover proofs more quickly, leading to a stronger unified strategy within the one-minute time limit.

6 Examples

Besides the example in Section 1, we present two more examples of the invented strategies that (dis)prove problems unprovable by any participant in CoCo.

The core structure of the first example is `AT`. It proves confluence for `794.ari` in ARI-COPS (`939.trs` in COPS). The sub-strategy `AT`, denoting Aoto-Toyama criteria [Aoto and Toyama, 2012], is defined in CSI's competition configuration document. CSI's competition strategy executes `AT` in parallel with many other sub-strategies, reducing the computational resources allocated to it and failing to find a proof.

Another example is similar to that in Section 1, we discover that if CSI employs `redundant -development 6` to generate redundant rules in the competition strategy, it can disprove confluence for `852.ari` (`997.trs` in COPS), and the proof can be certified by CeTA.

7 Related Work

There have been several attempts to apply machine learning to rewriting; however, none have been applied to automatic confluence provers. While [Winkler and Moser, 2019] investigate feature characterization of term rewrite systems, they do not build any learning models based on the features. There are works analyzing the termination of programs using neural networks to learn from the execution traces of the program [Giacobbe *et al.*, 2022; Abate *et al.*, 2021]. Nevertheless, they do not transform programs to term rewrite systems and apply machine learning to guide automatic term rewriting tools in termination analysis. MCTS-GEB [He *et al.*, 2023] applies reinforcement learning to build equivalence graphs for E-graph rewriting, but it focuses on optimization problems, not on confluence.

There has been extensive research on parameter tuning and strategy portfolio optimization in automated reasoning. Hydra [Xu *et al.*, 2010] employs a boosting algorithm [Freund and Schapire, 1997] to select complementary strategies for SAT solvers. [Ramírez *et al.*, 2016] propose an evolutionary algorithm for strategy generation in the SMT solver Z3 [De Moura and Bjørner, 2008]. A comprehensive review of these approaches is provided by [Kerschke *et al.*, 2019].

8 Conclusion and Future Work

We have proposed an approach to automatically invent strategies for the state-of-the-art confluence analysis prover CSI. We have performed data augmentation by randomly generating a large number of term rewrite systems and mixing these with the human-built dataset ARI-COPS. We have evaluated the invented combined strategy both on the original ARI-COPS dataset and the augmented dataset. The invented strategies discover significantly more proofs than CSI's competition strategy on both datasets. Notably, five of the human-written problems have never been proved by any automatic confluence provers in the annual confluence competitions.

Future work includes applying machine learning to individual term-rewriting techniques, for example those that perform search in a large space. Prioritizing the more promising parts of the search space could improve the individual techniques. Our strategy invention approach could also be extended to other automatic term rewriting provers. It would also be possible to apply neural networks to directly predict appropriate strategies for automatic term rewriting tools, however, soundness of proofs generated using such an approach remains a major challenge.

Acknowledgements

This research was supported by the ERC PoC project *Formal-Web3* no. 101156734, the University of Innsbruck doctoral scholarship *promotion of young talent*, the National Natural Science Foundation of China 92370201, and the Czech Science Foundation project no. 24-12759S.

References

[Abate *et al.*, 2021] Alessandro Abate, Mirco Giacobbe, and Diptarko Roy. Learning probabilistic termination proofs. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33*, pages 3–26. Springer, 2021.

[Aleksandrova *et al.*, 2024] Kristina Aleksandrova, Jan Jakubuv, and Cezary Kaliszyk. Prover9 unleashed: Automated configuration for enhanced proof discovery. In *Proceedings of 25th Conference on Logic for Pro*, volume 100, pages 360–369, 2024.

[Aoto and Toyama, 2012] Takahito Aoto and Yoshihito Toyama. A reduction-preserving completion for proving confluence of non-terminating term rewriting systems. *Logical Methods in Computer Science*, 8, 2012.

[Baader and Nipkow, 1998] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1998.

[Bachmair and Ganzinger, 1994] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[Brown and Kaliszyk, 2022] Chad E Brown and Cezary Kaliszyk. Lash 1.0 (system description). In *International Joint Conference on Automated Reasoning*, pages 350–358. Springer, 2022.

[De Moura and Bjørner, 2008] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[Freund and Schapire, 1997] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[Giacobbe *et al.*, 2022] Mirco Giacobbe, Daniel Kroening, and Julian Parsert. Neural termination analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 633–645, 2022.

[Gramlich, 1996] Bernhard Gramlich. Confluence without termination via parallel critical pairs. In *Colloquium on Trees in Algebra and Programming*, pages 211–225. Springer, 1996.

[He *et al.*, 2023] Guoliang He, Zak Singh, and Eiko Yoneki. Mcts-geb: Monte carlo tree search is a good e-graph builder. In *Proceedings of the 3rd Workshop on Machine Learning and Systems*, pages 26–33, 2023.

[Hůla and Jakubův, 2022] Jan Hůla and Jakubův. Targeted configuration of an smt solver. In *International Conference on Intelligent Computer Mathematics*, pages 256–271. Springer, 2022.

[Hutter *et al.*, 2009] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of artificial intelligence research*, 36:267–306, 2009.

[Kerschke *et al.*, 2019] Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary computation*, 27(1):3–45, 2019.

[Lindauer *et al.*, 2022] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022.

[McCune, 2005] W. McCune. Prover9 and Mace4. http://www.cs.unm.edu/~mccune/prover9/, 2005. Accessed: 2025-05-21.

[Meseguer, 2003] José Meseguer. Software specification and verification in rewriting logic. *Nato Science Series Sub Series III Computer and Systems Sciences*, 191:133–194, 2003.

[Nagele *et al.*, 2015] Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp. Improving automatic confluence analysis of rewrite systems by redundant rules. In *26th International Conference on Rewriting Techniques and Applications (RTA 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[Nagele *et al.*, 2017] Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp. Csi: New evidence–a progress report. In *International Conference on Automated Deduction*, pages 385–397. Springer, 2017.

[Ramírez *et al.*, 2016] Nicolás Gálvez Ramírez, Youssef Hamadi, Eric Monfroy, and Frédéric Saubion. Evolving smt strategies. In *2016 IEEE 28th International Conference on Tools with Artificial Intelligence*, 2016.

[Sternagel and Thiemann, 2014] Christian Sternagel and René Thiemann. The certification problem format. *arXiv preprint arXiv:1410.8220*, 2014.

[Stump *et al.*, 2014] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: A cross-community infrastructure for logic solving. In *International joint conference on automated reasoning*, pages 367–373. Springer, 2014.

[Thiemann and Sternagel, 2009] René Thiemann and Christian Sternagel. Certification of termination proofs using ceta. In *International Conference on Theorem Proving in Higher Order Logics*, pages 452–468. Springer, 2009.

[Van Oostrom, 1994] Vincent Van Oostrom. Confluence by decreasing diagrams. *Theoretical computer science*, 126(2):259–280, 1994.

[Van Oostrom, 1997] Vincent Van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159–181, 1997.

[Willsey *et al.*, 2021] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[Winkler and Moser, 2019] Sarah Winkler and Georg Moser. Smarter features, simpler learning? In *Proceedings of the Second International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements*, 2019.

[Xu *et al.*, 2010] Lin Xu, Holger Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pages 210–216, 2010.

[Zantema, 2004] Hans Zantema. Relative termination in term rewriting. In *WST'04 7th International Workshop on Termination*, page 51, 2004.